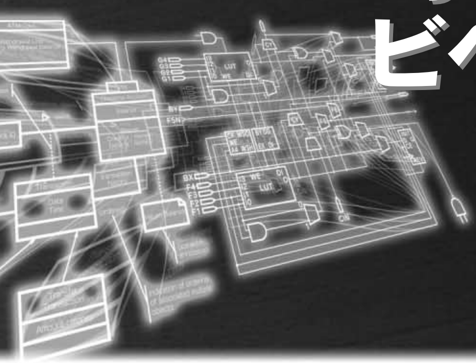
第3章

JPEGエンコーダのSystemCモデルを作成してビヘイビア合成

――ツールにまかせる部分と人手で最適化する部分を見極める



赤星博輝

ここでは，ディジタル・カメラのJPEGエンコーダの開発をイメージしながら，SystemCとビヘイビア合成の利用方法を解説する．JPEGエンコーダの処理のうち，ハードウェア化したときの効果が高いと思われるRGB-YUV変換とDCT（離散コサイン変換）の部分についてSystemCモデルを作成し，ビヘイビア合成を実施した．ビヘイビア合成ツールにすべてをまかせることは現実的ではなく，人手による最適化を加えて性能を逐次改善する必要がある．（編集部）

最近のLSI設計は規模に対して設計期間が短いことが多いのですが，今回の記事も設計期間（と執筆期間）がたいへん短く，その制約の中でビヘイビア合成を用いてどのようにハードウェア化を行ったかについて紹介します．

設計期間が短いことから，設計の手戻りを極力避ける必要があり，最初に設計フローを検討しました．検討した設計フローでは，まず全体の分析を行うためにSystemCでモデリングし，次にそのモデルを分析してビヘイビア合成を行う場所を決定し，続いてビヘイビア合成用のSystemCモデルを作成します（**図1**）．ビヘイビア合成用モデルの記述を，ただ合成可能なだけのモデルから，よりサイクル数の短い回路を生成するモデルへ書き換えました．その後，ビヘイビア合成を行った結果をフィードバックした分析モデルを作成し，少し詳細な評価を行いました．

ここではビヘイビア合成ツールとして，礎デザインオートメーションの「DesignPrototyper」を利用していますが，最適化の考えかたなどは，ほかのツールでも共通に適用できると思います．

● JPEGエンコーダ部を対象に設計

今回，ハードウェア化を検討したのはDCT（離散コサイン変換）を基本としたJPEG（Joint Photographic Experts Group）のエンコーダ部です（p.52のコラム「DCTとは」を参照）．JPEGは画像圧縮/伸張方式の一つで，非可逆変換と可逆変換，およびモノクロとカラーに対応しています．

処理概要を**図2**に示します．入力された画像がRGBで与えられます．これをYUV（YC_bC_r）に変換し，その変換されたYUVに対してDCT処理を行い，周波数空間に変換します（p.54のコラム「RGBとYUV」を参照）．その後，量子化処理で信号レベルを制限し，ハフマン符号化を行います．量子化処理では，低い周波数の信号は詳細に量子化し，高い周波数の信号は粗く量子化することで圧縮率を上げています．また，ハフマン符号ではよく出現する符号に短いコードを，あまり出現しない符号に長いコードを割り当てる

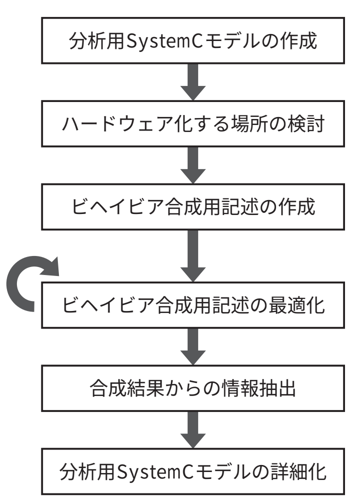


図1　今回の設計フロー
複数のことを同時に行うと失敗する可能性が高く，失敗したときの解析に時間がかかる．「急がば回れ」の発想で設計フローを考えた．

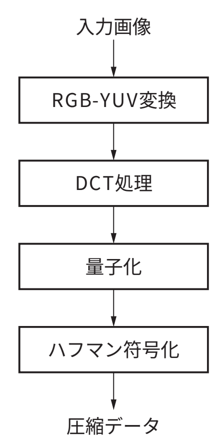


図2　JPEGエンコーダの概要
入力画像のRGBデータをYUV（YC_bC_r）データに変換する．その変換されたYUVデータに対してDCT処理を行い，周波数空間に変換する．その後，量子化処理で信号レベルを制限し，ハフマン符号化を行う．

3

ことで圧縮率を高めています．圧縮データから画像を生成するには，圧縮と逆の手順で実行することになります．

設計フローの第1段階として，ここではJPEGエンコーダ（以下，JPEG）のSystemCモデルを作成します．といってもJPEGのプログラムを一から作るとたいへんです．SystemCを使う利点の一つは既存のコードを再利用しやすいことなので，この部分はサボることを考えました．つまり，すでにC言語で作成されたJPEGのプログラムをリファレンスとして利用します．検索サイトで「JPEG　ソース」などと入力して検索すると，Independent JPEG GroupのWebサイト（http://www.ijg.org/）にたどり着きました．その使用条件を見たところとくに問題がないと思われるので，ftp://ftp.uu.net/graphics/jpeg/jpegsrc.v6b.tar.gzのソース・コードを利用しました（利用などに際しての条件は，同梱のREADMEファイルの記述に従うものとする）．

● リファレンスCコードからSystemCモデルを作成

このようなリファレンスのC/C++プログラムがあれば，第1段階のSystemCモデルの作成は簡単になります．このリファレンスからSystemCモデルを作成するため，以下の作業を行う必要がありました．

リスト1
JPEGのSC_MODULEの定義
起動のイベントとなるcmdを定義し，リファレンスのmain関数を利用するためにコンストラクタでSC_THREADのプロセスmainを宣言した．

```
SC_MODULE(jpeg6)
{
  sc_fifo_in<int> cmd;

  void main();

  SC_CTOR(jpeg6) {
    SC_THREAD(main);
    }
};
```

1）SC_MODULE（jpeg6）の作成

まずは，SC_MODULEとしてjpeg6を作成します（**リスト1**）．この段階ではクロックやリセットを意識しないので，プロセスとしてSC_THREADを選択します．コンストラクタにSC_THREADとしてmainを宣言しました．JPEGの処理を起動する信号が必要になるため，入力信号cmdを作成しました．**リスト1**を見てわかるように，簡単にSC_MODULEを定義できます．

2）main関数の変更

さらにプロセスmainを完成させる必要があります．ここではリファレンスのmain関数を変更し，SystemCのプロセスにしました．これはリファレンスのC言語記述がSystemCの流儀に従っていないためで，以下の2種類の変更が必要でした．

- **int main（int argc，char **argv）の返り値，引き数の変更**
- **プロセスにおけるwhileループの作成**

リスト2に示すように，モジュールのmainプロセスにするため，返り値をvoidとし，引き数をローカル変数にしました．また，引き数をローカル変数にすると必要な情報が渡せなくなるので，グローバル変数を利用してデータを渡すことにしました．あまり美しいとは言えない方法なのですが，時間をかけずにSystemCにする一つの方法ではあると思います．

リファレンスでは1度処理すると終了するプログラムとなっていましたが，今回の用途では要求があったときにJPEGの処理を行うことになるので，mainの処理部分をwhile(1)で囲みます．

COLUMN　DCTとは

画像処理では，離散フーリエ変換（DFT：discrete Fourier transform）の一種である離散コサイン変換（DCT：discrete cosine transform）がよく利用されます．とくに8画素×8画素のブロックを対象としたDCTがよく用いられ，これは2次元の画像信号を周波数空間に変換することに相当します．また逆に，周波数空間から画像信号に戻す変換はIDCT（inverse DCT；逆離散コサイン変換）と呼ばれています．

8画素×8画素のDCTは以下のように定義されます．

$$DCT(i,j)=\frac{1}{4}C_iC_j\sum_{x=0}^{7}\sum_{y=0}^{7}pixel(x,y)\cos\frac{(2x+1)i\pi}{16}\cos\frac{(2y+1)j\pi}{16} \quad \cdots\cdots\cdots(a\text{-}1)$$

ここで，$i=0$のときは$C_i=1/\sqrt{2}$，そのほかのときは$C_i=1$になります．

DCTが画像処理によく使われる理由は，以下のとおりです．

- 高速な解法（chen，wangなど）があり，画像のようなサイズの大きいデータにも適用できる
- 整数演算で実現できるため，（浮動小数点演算を使う場合より）容易に利用できる
- 高域と低域を別々に圧縮することにより，視覚的に劣化の少ない画像圧縮を実現できる（自然画像では低域の成分が大きくなり，高域の成分が小さくなる傾向にあり，かつ人間の感覚は高域より低域の変化に敏感であるため）

SC_THREADでは，JPEGの処理要求があったときだけ処理を行いたいので，そのwhile文の直後にsc_fifo_inの入力cmdを読むようにすることで，外部からのコマンド入力を待ちます．これで完成したかと思ってコンパイルして実行すると，本来，複数回JPEGエンコードを行うはずなのに，1回実行したところで終了してしまいました．よくプログラムを見てみると，リファレンスのプログラムの最後に処理が終了すると，exit関数を呼んで終了していました．今回のモデルではexitせずに次の処理を待つ必要があるので，その部分をコメント・アウトしました．

これにより，実行可能な（つまり，シミュレーション可能な）SystemCのモデルを10分程度で作成することができました．画像の入力・出力にファイルを使うことになりますが，ほかのモジュールとデータのやり取りを行うことも可能です．ファイル渡しではなく，メモリのモジュールなどを作って入力・出力できるようにすると，もう少し実機に近いモデルを作成できます．この第1段階では，そのような詳細化を行う前に，ここで作成したSystemCモデルを使って，ハードウェア化するためのJPEGのプログラムの挙動を調べることにします．

リスト2　C言語からSysetmCへの変更作業

main関数をSystemCのプロセスに変更する．そのために，引き数をローカル変数とし，グローバル変数から値をコピーする．また，while(1)で囲むことにより，繰り返し処理できるようにする．

```
int
main (int argc, char **argv)
{
  //  変数定義以下省略
```

（a）Cプログラムのmain関数

```
char  global_inputfile[256];
char  global_outputfile[256];

void
jpeg6::main ()
{
     int argc;
     char argv[100][256];
    /*  残りの変数定義省略*/
  while(1) {
     cmd.read();

   argc =6;
     argv[0]=”cjpeg”;
     argv[1]=”-outfile”;
     strcpy(global_outputfile, global_inputfile);
     strcat(global_outputfile, “.jpg”);
     argv[2]=global_outputfile;
     argv[3]=”-dct”;
     argv[4]=”fast”;
```

（b）SystemCのプロセスに変更

● プロファイルをとってハード化する箇所を決定

今回は設計期間が短いため，JPEGエンコーダ全体をハードウェア化することは困難でした．そこで，もっとも効率が良いと考えられる場所に絞ってハードウェア化することにしました．

みなさんは，どのようにしてハードウェア化する場所を決めますか？ 小規模な設計であれば，設計者の勘を信じて決めてもそれほど問題はありません．しかし，大規模な設計では，裏づけがなければ怖くて開発できないと思います（もし，予定の性能が出なかったらどうする？）．

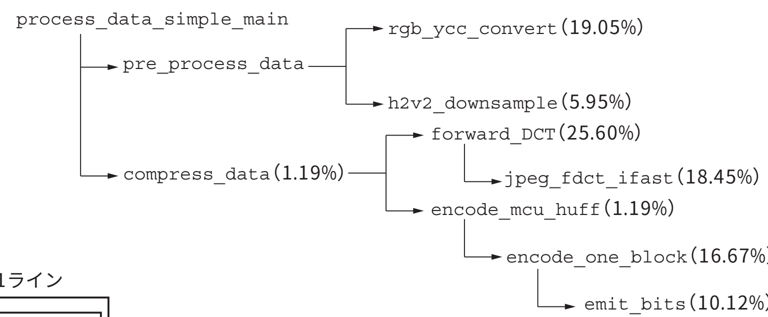
ここではソフトウェアの世界でよく利用されているプロファイラを利用します．SystemCを使うにはC/C++コンパイラが必要ですが，一般にそのコンパイル・オプションとしてプロファイラが用意されています．gccではオプション-pgを付けてコンパイルすると，プログラムの実行情報がgmon.outに出力され，gprofコマンドでプロファイル情報を見ることができます．

図3からわかるように，上位三つの関数でソフトウェアとしての実行時間の63%を超えています．このソフトウェアの実行時間とハードウェア化したときの実行時間は直接的な関係はないのですが，ビヘイビア合成を使う場合にはそれなりの相関があるとみなし，今回はこのプロファイル情報を利用することにしました．また，同じアルゴリズムを使えば，その呼び出し回数などの値はそのままハードウェアでも利用できます．

図3
プロファイラの結果（上位8関数）
プロファイル・オプションを付けてC/C++（SystemC）をコンパイルし，実行すると，各関数の実行時間の割合や呼び出し回数などを確認することができる．

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
25.60	0.04	0.04	21700	0.00	0.00	forward_DCT
19.05	0.07	0.03	2560	0.01	0.01	rgb_ycc_convert
18.45	0.11	0.03	32550	0.00	0.00	jpeg_fdct_ifast
16.67	0.13	0.03	32550	0.00	0.00	encode_one_block
10.12	0.15	0.02	584930	0.00	0.00	emit_bits
5.95	0.16	0.01	2560	0.00	0.00	h2v2_downsample
1.19	0.16	0.00	5425	0.00	0.01	encode_mcu_huff
1.19	0.16	0.00	160	0.01	0.77	compress_data

▶ **図4**
関数の呼び出し関係
ある程度のまとまりをハードウェア化すると作業が楽になる．



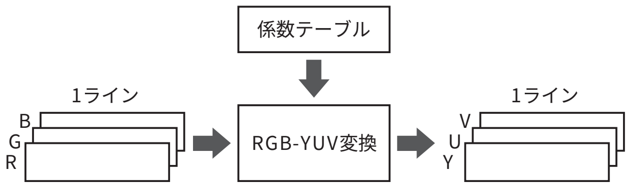


図5　RGB-YUV変換の入力および出力
リファレンスのCプログラムではもっと多くのデータにアクセスできるが，ハードウェア化する場合には，変換にほんとうに必要な入力と出力を明確にする必要がある．

今回は，上位の関数を中心にハードウェア化を検討することにしました．関数の呼び出しの関係を**図4**に示します．そしてその内部を調べて，`rgb_ycc_convert`とDCT（`forward_DCT`と`jpeg_fdct_ifast`）の部分をハードウェア化することに決定しました．

● RGB-YUV変換の処理をビヘイビア合成

次に，ビヘイビア合成を用いてどのようにハードウェア化したかについて説明していきます．

ビヘイビア合成ツールではSystemCの記述のすべてを合成できるわけではないので，合成可能な記述（合成サブセット）に変更する必要があります．先ほどリファレンスのCプログラムをSystemC化したプログラムも，そのままではビヘイビア合成できません．

RGB-YUV変換を行っている関数を**リスト3**に示します（オリジナルから少し変更している）．このままでは合成できないのですが，その大きな理由は，リファレンス記述がポインタを使ってアドレス計算を行い，配列にアクセスしているためです．そのようなプログラムは，今回のビヘイビア合成ツールでは合成できません．大きな構造体のポインタも渡しており，このポインタを使うと，もともと関数では使用しない情報などへもアクセスできます．しかし，ハードウェアではポート経由やメモリ経由でデータを転送するため，むだな情報を渡すと，むだな配線やむだな記憶

リスト3　RGB-YUV変換のリファレンス・コード

```
void  rgb_ycc_convert (j_compress_ptr cinfo,
                       JSAMPARRAY input_buf, JSAMPIMAGE output_buf,
                       JDIMENSION output_row, int num_rows)  {
  my_cconvert_ptr cconvert = (my_cconvert_ptr) cinfo->cconvert;
  register int r, g, b;
  register INT32 * ctab = cconvert->rgb_ycc_tab;
  register JSAMPROW inptr, outptr0, outptr1, outptr2;
  register JDIMENSION col;
  JDIMENSION num_cols = cinfo->image_width;

  inptr = *input_buf++;
  outptr0 = output_buf[0][output_row];
  outptr1 = output_buf[1][output_row];
  outptr2 = output_buf[2][output_row];
  output_row++;
  for (col = 0; col < num_cols; col++) {
    r = GETJSAMPLE(inptr[RGB_RED]);
    g = GETJSAMPLE(inptr[RGB_GREEN]);
    b = GETJSAMPLE(inptr[RGB_BLUE]);
    inptr += RGB_PIXELSIZE;
   outptr0[col] = (JSAMPLE)
      ((ctab[r+R_Y_OFF] + ctab[g+G_Y_OFF] + ctab[b+B_Y_OFF])
       >> SCALEBITS); /* Y */
   outptr1[col] = (JSAMPLE)
      ((ctab[r+R_CB_OFF] + ctab[g+G_CB_OFF] + ctab[b+B_CB_OFF])
       >> SCALEBITS); /* Cb */
   outptr2[col] = (JSAMPLE)
      ((ctab[r+R_CR_OFF] + ctab[g+G_CR_OFF] + ctab[b+B_CR_OFF])
       >> SCALEBITS);  /* Cr */
  }
}
```

COLUMN　RGBとYUV

光の3原色は赤（red），緑（green），青（blue）ですが，画像を圧縮する場合はRGBではなく，YUVを使います．ここでYは輝度信号，U（C_b）とV（C_r）は色差信号です．人間の目の特性として，輝度に比べて色差の変化に対する感度が鈍いため，色差信号は輝度信号より高い効率で圧縮することが可能です．

本稿では，RGBからYUVに変換する際に以下の式を使用しました．

$$Y = R\times0.2990 + G\times0.5870 + B\times0.1140$$
$$U = -R\times0.1684 - G\times0.3316 + B\times0.5000 + 128$$
$$V = R\times0.5000 - G\times0.4187 - B\times0.0813 + 128$$
……(b-1)

リスト4　RGB-YUV変換のモジュール宣言

RGBとYUVのデータを格納する配列をグローバル変数として宣言し，ほかのモジュールから読み書きできるようにした．

```
#include <systemc.h>
 unsigned char buf_R[2048];
 unsigned char buf_G[2048];
 unsigned char buf_B[2048];
 unsigned char buf_Y[2048];
 unsigned char buf_U[2048];
 unsigned char buf_V[2048];

SC_MODULE(rgb_yuv) {
    sc_in<bool >        clock;
    sc_in<bool >        reset;
    sc_in<bool >        start;
    sc_in<sc_int<16> >  width;
    sc_out<bool >       done;
    sc_int<16>          w;
                       // widthを記憶する変数
    void rgb_ycc_convert( int num_cols);
    void main_process();
    SC_CTOR(rgb_yuv) {
      SC_CTHREAD(main_process, clock.pos());
      watching(reset.delayed() == 1);
    }
};
```

リスト5　rgb_ycc_convertを合成可能に変更

ポインタを使用しない形に記述を変更し，係数テーブルを配列として宣言した．

```
sc_int<32> ctab[2048]={
/*中身は省略*/
}; // ida map_to_resource rom

#define SCALEBITS     16
void rgb_yuv::rgb_ycc_convert(int num_cols)  {
  int r, g, b;
  sc_int<16> col;
  char y,u,v;
  sc_int<32> ty, yu, tv;

  for (col = 0; col < num_cols; col++) {
    r = buf_R[col];
    g = buf_G[col];
    b = buf_B[col];
    ty =(ctab[r] + ctab[g+1*256] + ctab[b+2*256]) >> SCALEBITS;
    y = ty.range(7,0);
    buf_Y[col]=y;
    u= (ctab[r+3*256] + ctab[g+4*256] + ctab[b+5*256])>> SCALEBITS;
    buf_U[col]=u;
    v= (ctab[r+5*256] + ctab[g+6*256] + ctab[b+7*256])>> SCALEBITS;
    buf_V[col]=v;
  }
}
```

領域，むだな実行サイクルが発生してしまいます．

今回は，以下の手順でビヘイビア合成を行いました．

1）入力と出力を明確化

むだな情報を渡さないためにもRGB-YUV変換に必要なデータをはっきりさせます．

この機能に必要なものは，入力については1ライン分のRGBデータ，ライン・サイズ，係数の配列，出力については1ライン分のYUVデータとなります(**図5**)．今回は，最大のライン・サイズを2048とし，RGBおよびYUVのデータごとに配列(メモリ)を定義しました．ライン・サイズは，今後の拡張も含めて16ビットの整数としています．係数の配列については，内部にROMとして用意します．そのため，入出力から外しました．

また，ハードウェア化するときに必要な信号として，クロック(clock)，リセット(reset)，および処理の開始を指示するstart信号，処理の完了を示すdone信号を用意しました．

2）コンストラクタを作成

今回のツールでビヘイビア合成を行うためには，SC_CTHREADでプロセスを定義する必要があります．オリジナルの関数rgb_ycc_convertはSC_CTHREADとしては不十分なので変更せず，ここではmain_processというプロセスを用意し，その中からrgb_ycc_convertを呼び出すことにします．ここまでで，**リスト4**に示すように基本的なクラスを定義することができます．

main_processをSC_CTHREADとして宣言するとき，clock.pos()とすると，クロックの立ち上がりで動作することになります．また，watchingでresetが‘1’のときに同期リセットがかかることを宣言しています．

3）機能を合成可能な記述へ書き換え

リファレンスで記述されていたままのコードではビヘイビア合成できないので，ポインタなどを配列に書き換えます．また，係数テーブルの配列のデータもあらかじめ計算したものを用意します．その結果，rgb_ycc_convertは**リスト5**のように変わります．

また，係数の配列はROMとして実現するので，コメントでROMにするディレクティブを与えています．

4）初期化および処理手順を決定

リスト6にmain_processの処理を示します．ここでは初期化および処理手順を記述します．最初に出力信号doneを初期化します．その後，while(1)で繰り返し処理を行います．この繰り返しでは，start信号が‘1’になるまで待ちます．start信号が‘1’になれば，処理するライン・サイズを読み出し，rgb_ycc_convertを呼び出してRGB-YUV変換を行います．rgb_ycc_convertの処理が完了したら，done信号を‘1’にし，その次のサイクルでdone信号を‘0’にして，start信号が‘1’になるのを待ちます．

5）sc_mainを作成

リスト7に示すように，今回使用したビヘイビア合成ツ

リスト6　main_processの定義

```
void rgb_yuv::main_process()  {
  // initialize
  done.write(0);
  wait();

  while(1) {
   wait_until(start.delayed() == 1) ;
   w = width.read();
   rgb_ycc_convert(w);
   done.write(1);
   wait();
   done.write(0);
  }
}
```

リスト8　resource.resの内容

Altera社のライブラリlpm_1p_ram_8x2048**と今回用意した**myrom_32x2048**を定義している．**

```
$def_resfile ;

%mapmem -res lpm_1p_ram_8x2048 -var buf_R -inst umR;
%mapmem -res lpm_1p_ram_8x2048 -var buf_G -inst umG;
%mapmem -res lpm_1p_ram_8x2048 -var buf_B -inst umB;

%mapmem -res lpm_1p_ram_8x2048 -var buf_Y -inst umY;
%mapmem -res lpm_1p_ram_8x2048 -var buf_U -inst umU;
%mapmem -res lpm_1p_ram_8x2048 -var buf_V -inst umV;

%mapmem -res myrom_32x2048 -var ctab -inst uROMctab;


$end_def_resfile ;
```

ールでは，sc_main内にハードウェア化するモジュールをインスタンスし，ポート接続を行う必要があります．このとき，合成するモジュールにコメントでディレクティブをつけます．

6）ビヘイビア合成を実行

これでようやくビヘイビア合成が行えます．ここでは今回使用したビヘイビア合成ツールに付属するライブラリの関係から，米国Altera社のFPGA「FLEX10KEファミリ」を対象にビヘイビア合成を行いました．もちろん合成用ライブラリを用意すれば，ほかのデバイス（ASICやFPGA）をターゲットとしたRTL（register transfer level）記述も生成できます．

今回は，RGBとYUVのデータを外部メモリ（RAM），係数テーブルを内部メモリ（ROM）として実現します．今回のビヘイビア合成ツールでは，使用するRAM，ROMの情報を用意し，設計の中のどの配列とどのRAM，ROMが対応するかをresource.resに記述します（**リスト8**）．また，外部メモリとして実現する場合には，実行時にオプションで「-extmem　インスタンス名」を指定します．

リスト7　sc_mainの作成

今回使用したビヘイビア合成ツールはsc_main**を見て合成モジュールを決定するので，インスタンスし，ポート接続を行う必要がある．**

```
int sc_main(int argc, char **argv)  {
  sc_set_time_resolution(1, SC_NS);
  sc_set_default_time_unit(1, SC_NS);

  sc_clock clock(“clock”, 100, 0.5, 0, true);
  sc_signal<bool >         reset;
  sc_signal<bool >         start;
  sc_signal<bool >         done;
  sc_signal<sc_int<16> >   w;
  rgb_yuv u(“urgb_yuv0”);      // ida TYPE HP
  u.clock(clock);
  u.reset(reset);
  u.start(start);
  u.width(w);
  u.done(done);

  /* テストベンチ記述  */

}
```

7）RTLでシミュレーション

RTLシミュレーションを行って，問題がないかどうかをチェックします．基本的には，SystemCやC/C++のシミュレーション結果と一致するかどうかをチェックすることになります．すでにSystemCやC/C++のモデルがあるので，期待値を作成するのは簡単です．

本ツールはVerilog HDLのRTL記述を生成します．SystemCとVerilog HDLの混在シミュレーションが可能な検証環境を利用すれば，SystemCのモジュールとVerilog HDLのRTL記述を同時に動かしてチェックできます．また，SystemVerilogのDPI（direct programming interface）を使ってCプログラムを呼び出すことで，期待値を生成する方法も利用できます．

今回はシンプルに，RGB-YUVの入力値と期待値をファイルで与えることにします．そのため，SystemCでJPEGのエンコードを実行しているときに，RGBおよびYUVのデータを2進表示のテキスト形式のファイルに出力しました．C言語だと2進数に関するサポートがありませんが，SystemCは変換や出力フォーマットのサポートがあります．**リスト9**に示すように，今回はto_stringとhexを使いました．これにより，必要なデータを出力することができます．

● 性能改善のための最適化を行う

合成してシミュレーション結果が正しいことを確認したら，次は最適化を行います．RGB-YUV変換では，RGBの

リスト9 SystemCでダンプする方法

SystemCのデータ型と，to_stringやhexといった記述を利用すると，2進数や16進数でダンプしやすい．

```
sc_uint<8> tr;
tr = r;
cout <<tr.to_string(SC_BIN,0) << “
               // 0x”  << hex << tr << “¥n”;
```

リスト10 ROMを複数に分割

ROMを九つに分割したため，係数を同時に読み出せる．

```
int    myctabYR[256]={/*省略*/};// ida map_to_resource rom
int    myctabYG[256]={/*省略*/};// ida map_to_resource rom
int    myctabYB[256]={/*省略*/};// ida map_to_resource rom
int    myctabUR[256]={/*省略*/};// ida map_to_resource rom
int    myctabUG[256]={/*省略*/};// ida map_to_resource rom
int    myctabUB[256]={/*省略*/};// ida map_to_resource rom
int    myctabVR[256]={/*省略*/};// ida map_to_resource rom
int    myctabVG[256]={/*省略*/};// ida map_to_resource rom
int    myctabVB[256]={/*省略*/};// ida map_to_resource rom

void rgb_yuv::rgb_ycc_convert(int num_cols)  {
  int r, g, b;
  sc_int<16> col;
  char y,u,v;
  sc_int<32>  ty,  tu, tv;
  for (col = 0; col < num_cols; col++) {
    r = buf_R[col];
    g = buf_G[col];
    b = buf_B[col];
    ty = myctabYR[r]  + myctabYG[g]  + myctabYB[b];
    y = ty >> SCALEBITS;
    buf_Y[col]=y;
    tu =myctabUR[r]+ myctabUG[g]+ myctabUB[b];
    u= tu >>SCALEBITS;
    buf_U[col]=u;
    tv = myctabVR[r]  + myctabVG[g]+ myctabVB[b];
    v=  tv >>SCALEBITS;
    buf_V[col]=v;
  }
}
```

三つの値を処理するために，9個の係数を一つのROMから読み込んでおり，このアクセスが性能向上のボトルネックになっていました．Cベース設計では配列を使って処理を記述することが多いのですが，ここではこの問題の解決方法を三つ紹介します．

- **方法1：一つのROMを9個のROMに分割し，同時にアクセスできるようにする**

これは**リスト5**のctabで定義した2048要素の配列を，**リスト10**のように9分割して，256要素の配列にする方法です（ctabUBとctabVRは同じ内容だが，同時にアクセスするために個別に用意している）．

- **方法2：前述の9個のROMの内容を組み合わせ回路で実現する**

あらかじめ用意したVerilog HDLのモジュールを利用してビヘイビア合成を行います．そのため，**リスト11**に示すように関数を定義し，Verilog HDLのモジュールとの関連を示すアトリビュートを付けておきます．

- **方法3：生成アルゴリズムを組み合わせ回路で実現する**

今回の係数テーブルは，定数*indexという比較的簡単な形で定義できます．**リスト12**に示すように，計算式を

リスト11 ROMの内容を組み合わせ回路で実現

九つに分割したROMに相当する組み合わせ回路をHDLで作成し，ビヘイビア合成でその組み合わせ回路を利用した．

```
SC_MODULE(rgb_yuv) {
  ・・・
   /* 追加  */
    void  func_YR(int i, int *o) ;
    void  func_YG(int i, int *o) ;
    void  func_YB(int i, int *o) ;
    void  func_UR(int i, int *o) ;
    void  func_UG(int i, int *o) ;
    void  func_UB(int i, int *o) ;
    void  func_VR(int i, int *o) ;
    void  func_VG(int i, int *o) ;
    void  func_VB(int i, int *o) ;
  /*追加終わり*/
  ・・・
};

void  rgb_yuv::func_YR(int i, int *o)
  /* ida map_to_module ctab_YR
     input ina(i):32,  output=res(o):32
   */
{
  *o = tbl_YR(i);
}
```

（注記）九つのROM相当の機能をSystemCの関数として実現する

（注記）Verilog HDLのモジュールctab_YRにマッピング．ctab_YRのinaにiが対応し，resにoが対応する

リスト12 生成アルゴリズムを組み合わせ回路で実現

今回の係数テーブルは比較的単純な計算で生成されており，直接その生成アルゴリズムを関数（マクロ）として実現した．

```
#define  tbl_RY(i)  (FIX0_299*i)
#define  tbl_RG(i)  (FIX0_587*i)
#define  tbl_RB(i)  (FIX0_114*i+FIX_ONEHALF)
#define  tbl_UR(i)  (-FIX0_168*i)
#define  tbl_UG(i)  (-FIX0_331*i)
#define  tbl_UB(i)
             (FIX0_500*i+FIX_CBCR_OFF+FIX_ONEHALF-1)
#define  tbl_VR(i)  (tbl_UB(i))
#define  tbl_VG(i)  (-FIX0_418*i)
#define  tbl_VB(i)  (-FIX0_081*i)

void rgb_yuv::rgb_ycc_convert(int num_cols)  {
  int r, g, b;
  sc_uint<16> col;
  unsigned char y,u,v;
    for (col = 0; col < num_cols; col++) {
      r = buf_R[col];
      g = buf_G[col];
      b = buf_B[col];
      y= ((tbl_RY(r)  + tbl_RG(g)  + tbl_RB(b)) >>
                                         SCALEBITS);
      buf_Y[col]=y;
      u= ((tbl_UR(r)+ tbl_UG(g)+ tbl_UB(b))>>
                                         SCALEBITS);
      buf_U[col]=u;
      v= ((tbl_VR(r)  + tbl_VG(g)+ tbl_VB(b))>>
                                         SCALEBITS);
      buf_V[col]=v;
    }
}
```

3

使ってRGB-YUV変換を行えます.

合成結果として，初期解，方法1，方法2，方法3の各実行サイクルを**表1**に示します．どの手法が良いかはケース・バイ・ケースです．ビヘイビア合成を利用する利点の一つは，このように短時間で複数のアーキテクチャを比較・検討できることです．

なお，今回，最適化の作業を行っていたときに，ビヘイビア合成によって生成されたRTL記述とSystemCモデルのシミュレーション結果が一致しないケースが見つかりました．原因を追いかけてみると，データを保持するためのビット幅が不足しているところがありました．このようなときは，計算結果が正しいビット幅になるように中間変数（例えば，16ビットならsc_int<16>など）としたり，方法2で述べたHDLのモジュールを利用してビヘイビア合成を行うなどの手法が有効です．

● DCT処理を合成，最適化

今回，DCTはMCU（minimum coded unit）単位で行うことにしました．JPEGでは処理単位をMCUと呼びます．今回はY成分が4枚，U成分が1枚，V成分が1枚という構成です（1枚は8画素×8画素のブロックを指す）．

	サイクル数
最初の実装	1760
方法1：ROMを複数個使用	680
方法2：モジュールを使用	320
方法3：生成関数を使用	320

表1　RGB-YUVの実行サイクル数（160画素）

最適化していない記述と最適化した記述では実行サイクル数に大きな差が出る．ビヘイビア合成では，ツールが最適化する部分と設計者が最適化する部分をきちんと理解しなければならない．

リスト13　DCTのモジュール定義

```
int   xINP_buffer[6][64];

SC_MODULE(dct) {
  sc_in<bool >        clock;
  sc_in<bool >        reset;
  sc_in<bool >        start;
  sc_out<bool >       done;

  int RIGHT_SHIFT(int x,int shft);
  int  DESCALE(int x, int n);
  void jpeg_fdct_islow (int);
  void main_process();
  SC_CTOR(dct) {
    SC_CTHREAD(main_process, clock.pos());
    watching(reset.delayed() == 1);
  }
};
```

まずは，リファレンスをもとにして，RGB-YUV変換と同じようにSC_MODULEを作成します．**リスト13**に示すように，DCTのデータは入力，出力ともにxINP_bufferを利用し，このほかにクロック，リセット，start信号，done信号を用意しました．今回の合成にはjpeg_fdct_islowを使用しています．また，今回使用したビヘイビア合成ツールでは，合成対象のモジュール内で使用する関数はモジュールのメンバ関数にする必要があります．そのため，RIGHT_SHIFT，DESCALE，jpeg_fdct_islowの三つを定義しています．

main_processでは，**リスト14**に示すように初期化を行い，while(1)で無限ループを作成してMCU単位に処理を行います．そこで，6ブロックに対してDCTを実行する記述を作成しました．また，ポインタ変数およびアドレス計算を行っていた部分を，配列とそのインデックスに変更しました．

DCTの最適化は，以下の手順で行いました．

1）配列アクセスを人手で修正

今回使用したビヘイビア合成ツールでは，配列へのアクセスはメモリ（レジスタ・ファイル）へのアクセスとして実現されます．いったんメモリ・アクセスとして実現されてしまうと，それ以上の最適化は困難となり，性能（スループットやレイテンシ）向上のボトルネックになってしまいます．とくに，むだなメモリ・アクセスは減らすことが重要です．

今回のDCTの処理では，入力時に同じアドレスに2回ずつアクセスする記述になっています．ツールは記述のとおりに合成しており，この部分を人手で最適化してやることで，サイクル数を削減できると予想できます．そこで，1回だけアクセスする記述に変更したところ，約3割もサイクル数を削減できました．かなり大きな最適化と言えます．

リスト14　DCTのmain_processの定義

```
void dct::main_process() {
  int i;
    // initialize
    done.write(0);
    wait();
  while(1) {
    wait_until(start.delayed() == 1) ;
    for(i=0; i < 6 ; i++ )
jpeg_fdct_islow(i);
    done.write(1);
    wait();
    done.write(0);
  }
}
```

表2　DCTの実行サイクル数
8画素×8画素のブロックを6個処理するサイクル数を示した．

	サイクル数
初期解	2311
配列アクセスを人手で修正	1687
関数をHDLモジュールへマッピング	1543
データをパッキング	1207

表3
ハードウェア部の処理時間
wait で遅延時間を入れたSystemCモデルを利用することにより，より詳細な評価を実施することができる．

画像サイズ	処理時間（ms）
160 × 160	6.49
320 × 240	19.48
640 × 480	77.93
1280 × 960	311.73

2）関数をHDLモジュールへマッピング

次に，内部で呼び出しているDESCALEをHDLで設計したモジュールに置き換えます．DESCALEはさらにRIGHT_SHIFTを呼び出しているので，2個の関数を一つのHDLのモジュールに置き換えます．これによって，さらに2割，サイクル数を削減できました．

3）データをパッキング

最後に，配列のアクセスをもう少しくふうし，32ビット幅にパックすることで，一度に複数個のデータをやり取りできるようにしました．入力の配列は8ビット，出力の配列は16ビットあればよいため，入力では4個のデータを集めて32ビットとし，出力では2個のデータをまとめて32ビットとしました．これにより，さらに2割のサイクルを削減できました．

表2に，最適化によるサイクル数の変化を示します．初期解と比較すると，データをパッキングしたものはサイクル数を約50%に削減できています．これは，かなり大きな改善と言えます．

● ハードウェア部の処理時間を評価

ビヘイビア合成が終了し，RGB-YUVとDCTの二つのモジュールのRTLが完成しました．もとのプロファイルをとったSystemCモデルを使って，より詳細な評価を行ってみたいと思います．

評価を行う手法として，Verilog HDLとSystemCの混在シミュレーションを行う方法や，合成ツールが出力したVerilog HDLコードをSystemCに変換するツールを使う方法などがありますが，このほかに，実行時間をwait文で埋め込んでしまうという方法もあります．ビヘイビア合成では，入力されたSystemCモデルの機能と出力されたRTL記述の機能は同じです（ツールにバグがなければ…）．そこで，wait文で遅延を入れても動作は変わらない一方で，実行時間の評価はかなり正確に行えます．

RGB-YUVの実行時間は3×ライン・サイズ×クロック周期，1ブロックのDCTの実行時間は200×クロック周期として見積もることができます．この二つは，SystemCでは以下のように書くことができます．

```
wait(3*num_cols*CYCLE_TIME, SC_NS);
wait(200*CYCLE_TIME,SC_NS);
```

さらにコンパイルし，実行すると，**表3**のような見積もりを得ることができます．ハードウェア化した部分は正しい値なので，かなり精度の高い見積もりだと思います．ただし，今回はソフトウェアについての時間を入れていません．つまり，この時間だけでJPEGのエンコード処理が完了するわけではありません．

また，このモデルでは逐次処理で見積もりを行っています．機能の一部をハードウェア化すると，ソフトウェアの部分とハードウェアの部分を同時並行に動作させることができるため，同じ時間でより多くの処理を行えます．そして，並列に動作するハードウェアを増やすことで，さらに処理を高速化できます．こうした評価を正確に行うためには，SystemCのモデルを変更・修正していく必要があります．

うまくモジュール化し，インターフェースをきちんと定義することにより，より詳細なレベルで分析・評価できるようになりますし，再利用性が高いモデルを作ることもできます．とくにSystemCではトランザクション・レベル・モデリング（TLM）が今後重要となってきます．合成の面から見ると，ビヘイビア合成ツールがどこまでの記述スタイルに対応できるかなどが，今後の課題となりそうです．

あかぼし・ひろき
（株）ロジック・リサーチ

<筆者プロフィール>
赤星博輝．ロジック・リサーチに所属しており，ハードウェアからソフトウェアまで担当するなんでも屋．もともとはEDAツールの開発を行っていたが，現在はハードウェア設計から徐々に検証技術に移っているところ．今後，どのようなしごとをするのかは，世の中の進みかたしだいです．